PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : **H04Q** | **A2**

(11) International Publication Number: **WO 97/22208**

(43) International Publication Date: 19 June 1997 (19.06.97)

(21) International Application Number: PCT/GB96/02961

(22) International Filing Date: 29 November 1996 (29.11.96)

(30) Priority Data:

9525214.4	9 December 1995 (09.12.95)	GB
9600681.2	12 January 1996 (12.01.96)	GB

(71) Applicant *(for all designated States except US)*: NORTHERN TELECOM LIMITED [CA/CA]; World Trade Center of Montreal, 8th floor, 380 St. Antoine Street West, Montreal, Quebec H2Y 2Y4 (CA).

(71) Applicant *(for US only)*: NICHOL, John (Heir of HOWARD, Martyn (deceased)) (executor for the deceased inventor) [GB/GB]; 6 Barn Court, Station Road, Sawbridgeworth, Herts CM21 9QN (GB).

(72) Inventor: HOWARD, Martyn (deceased) (deceased).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: TURRELL, Stephen [GB/GB]; 32 Park Court, Harlow, Essex CM20 2PY (GB). NOKES, William [GB/GB]; 35 Milton Street, Waltham Abbey, Essex EN9 1EZ (GB).

(74) Agent: RYAN, John, Peter, William; Nortel Patents, London Road, Harlow, Essex CM17 9NA (GB).

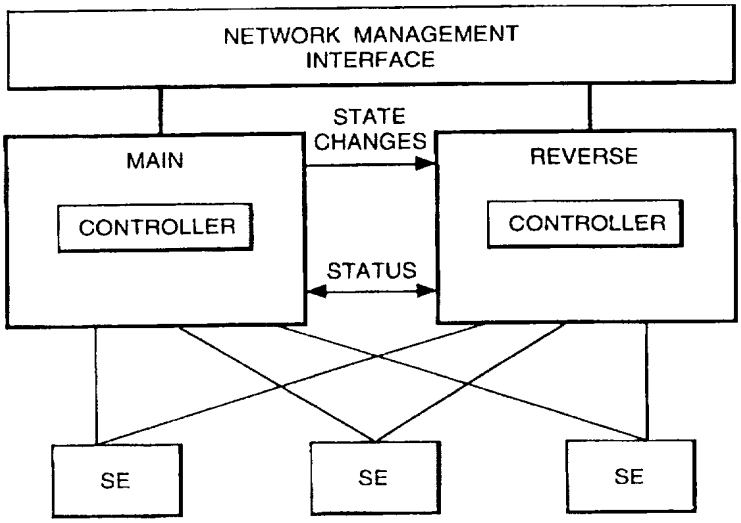
(81) Designated States: JP, US, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: PROVIDING ACCESS TO SERVICES IN A TELECOMMUNICATIONS SYSTEM

(57) Abstract

A communications network, includes a plurality of switch elements, a controller for the switch elements, and a system manager. The controller comprises a main computer system and a reserve computer system, there being means for switching from the main computer system to the reserve computer system in the event of failure of the main computer system. The main computer system communicates state changes in the form of a message stream to the reserve computer system whereby to update the reserve system with the current network state. The main computer system also provides status information to the reserve computer system at regular intervals whereby to detect an in-service failure of the main computer system.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	**GB**	United Kingdom	**MW**	Malawi
AT	Austria	**GE**	Georgia	**MX**	Mexico
AU	Australia	**GN**	Guinea	**NE**	Niger
BB	Barbados	**GR**	Greece	**NL**	Netherlands
BE	Belgium	**HU**	Hungary	**NO**	Norway
BF	Burkina Faso	**IE**	Ireland	**NZ**	New Zealand
BG	Bulgaria	**IT**	Italy	**PL**	Poland
BJ	Benin	**JP**	Japan	**PT**	Portugal
BR	Brazil	**KE**	Kenya	**RO**	Romania
BY	Belarus	**KG**	Kyrgystan	**RU**	Russian Federation
CA	Canada	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SG**	Singapore
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovakia
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LR**	Liberia	**SZ**	Swaziland
CS	Czechoslovakia	**LT**	Lithuania	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**LV**	Latvia	**TJ**	Tajikistan
DK	Denmark	**MC**	Monaco	**TT**	Trinidad and Tobago
EE	Estonia	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**UG**	Uganda
FI	Finland	**ML**	Mali	**US**	United States of America
FR	France	**MN**	Mongolia	**UZ**	Uzbekistan
GA	Gabon	**MR**	Mauritania	**VN**	Viet Nam

PROVIDING ACCESS TO SERVICES IN A TELECOMMUNICATIONS SYSTEM

This invention relates to the delivery of services to users of a telecommunications system.

Telecommunications networks were originally developed to provide voice communication between subscribers, this basic service now being referred to as POTS. With the introduction of digital networks it has become feasible to provide a variety of services to system users. Typically these services originate from a number of service providers, each service running on a software application which may be unique to that provider. This has introduced the problem of inter-operability and of providing a single means of access by a subscriber to those services. One approach to this problem is described by M Lapierre et al. in Electrical Communications, October 1994, pp260-7.

A further problem with advanced telecommunications networks is that of ensuring reliability by providing backup systems to take over in the event of a system fault so that a service to customers may be maintained. This is a particular problem with the network switches which, in a modern ATM network may comprise a lower switch element layer built in specialised hardware and providing the basic performance, a real time controller (RTC) providing a control functionality for the switch elements, and a system manager providing the network management interfaces. In order to meet reliability requirements, a number of key system components must be duplicated so that, in the event of a component failure, the duplicate can take over. At present, high reliability is provided by duplicating all hardware elements. This is costly both in

terms of hardware costs and development costs as such systems are proprietary in nature.

An object of the invention is to provide improved access by telecommunications subscribers to network services.

It as another object of the invention to provide an open distributed computer system associated with a telecommunications network.

It is a further object of the invention to provide a nucleus platform supporting a plurality of server interfaces.

According to one aspect of the invention there is provided a communications network, including a plurality of switch elements, a controller for the switch elements, and a system manager, wherein the controller comprises a main computer system and a reserve computer system, there being means for switching from the main computer system to the reserve computer system in the event of failure of the main computer system, wherein the main computer system has means for communicating state changes in the form of a message stream to the reserve computer system whereby to update the reserve system with the current network state, and wherein the main computer system is arranged to provide status information to the reserve computer system at regular intervals whereby to detect in service failure of the main computer system.

According to another aspect of the invention there is provided a method of controlling a telecommunications network including a plurality of switch elements, a controller for the switch elements, and a system manager, wherein the controller comprises a main computer system and a reserve computer system, the method including communicating state changes in the main computer system to the reserve computer system whereby to update the reserve system with the current network state, providing status information from the main computer system to the reserve computer system at regular intervals whereby to detect in service failure of the main computer system, and substituting the main

computer system with the reserve computer system in the event of a detected failure of the main system.

According to a further aspect of the invention there is provided a communications network providing client access to a plurality of servers, the network including a plurality of capsules each containing client and server objects and each incorporating a nucleus supporting a plurality of client/server interfaces, wherein each said nucleus includes means for pooling tasks, means for providing threads and for assigning those threads to respective tasks, and means for binding an assigned thread to its respective task during the execution of that task.

In our arrangement, the conventional tightly coupled duplicate hardware is replaced by two computing platforms, a master and a spare, which are loosely coupled via a network. The master handles all the work, with the spare monitoring activity and keeping in step so as to be ready to take-over in the event that the master fails. Obviously if the spare fails, the master continues to handle the system. This is cheaper in terms of hardware since no additional work is required to have the hardware self-check and manage switch over.

An embodiment of the invention will now be described with reference to the accompanying drawings in which:-

Figure 1 is a schematic diagram of a telecommunications network provided with duplication of control functions;

Figure 2 is a schematic diagram illustrating client server interaction via an open distributed system associated with a telecommunications network.

Figure 3 illustrates the general functionality of a nucleus of the system of figure 2;

Figure 4 shows the multithreading environment of the nucleus of figure 2; and

Figure 5 illustrates a thread/task model of the nucleus of figures 3 and 4.

Referring to figure 1, the network includes a plurality of switch elements (SE) which are used to determine traffic routing in the network. The switch elements are controlled via a real time controller and system management is provided by a network management interface (NMI) allowing an operator to monitor and configure the system. The controller functions are provided by a main computer and are duplicated or backed up by a reserve computer. In normal operation, the master provides all the system control functions, but, in the event of failure of the master, its function is replaced by the reserve. In this arrangement it is necessary to ensure that the reserve has up-to-date information about the state of the system, to detect failure and to take over control when the master fails. All of this is handled within a deterministic real-time computing environment. The first problem is handled by a process referred to as journalling. The master communicates state changes to the spare as a stream of messages. Failure detection is provided via a 'heartbeat' protocol wherein the master and reserve exchange status messages at regular intervals; a missing message denotes potential failure. The frequency of such heartbeats determines the maximum time taken to detect failure. Finally, switch over in the event of failure is managed by providing a path from each SE to both master and reserve and the SE is notified on failure to report via the backup path to the reserve.

Referring now to figure 2, the open distributed computer system incorporates a number of capsules 11 each containing client and server objects communicating via remote procedure calls (RPC) and each incorporating a nucleus 12. Communication between capsules is effected via telecommunications network 13. Within each capsule, the nucleus provides the infrastructure required for distribution, communication and multi-threading support for client and server objects. The nucleus further provides application concurrency. In the following description we define logical units of concurrency and physical unit, of concurrency and physical units of concurrency as thread and task respectively.

In particular:-

A logical unit of concurrency is an independent execution path through an application which can be executed concurrently with another typical unit of concurrency. A physical unit of concurrency is the resource required to enable execution of a logical unit of concurrency.

The nucleus is a modular design for real-time communications infrastructure which can be configured at compile and runtime to meet different domain requirements. Particular emphasis is on deterministic performance and resource utilisation. By providing a well-defined API covering all the necessary functionality, applications become portable and, by using the common infrastructure, are able to interwork regardless of the computing platform involved. The nucleus provides a set of well defined Application Programmer Interfaces and a strict modular approach allows alternative implementations of key components to be selected either at link or run time to meet particular application needs. By providing a common interface to the protocol stacks at the application level, the nucleus allows a number of separate protocol stacks to be utilitised simultaneously at run-time by application objects. The use of a wrapper which translates between protocol and Nucleus interface, allows most protocol stacks to be incorporated into the Nucleus communications framework. By providing a standard model of concurrency and resource usage, the application programmer can construct concurrent programs independently of the mechanisms by which the concurrency is actually provided. These can be supplied by the host Operating System (e. g POSIX Threads, or real-time kernel threads) or by proprietary components without changing the application code. This ensures that applications can have the flexibility of concurrency without sacrificing portability. At the same time, the model of resource usage ensures that the resources are controlled, prioritised and behave in a deterministic way. Within the concurrent environment, events and resources may be generated asynchronously and need handling according to priority. Again, the resources used for this

purpose must be controlled to ensure that priorities are honoured. To this end, the nucleus provides an asynchronous event notification mechanism and a flexible resource management system which combine to deliver the necessary control.

In a telecommunications environment, large numbers of events are occurring which are of interest only to parts of the system. In many cases, the system models are dependent on asynchronous events to progress activity (e. g. a standard call model). In order to support this, a mechanism to manage the raising, queuing and handling of events in a well-defined context with a view to priority and determinism is necessary. The Asynchronous Event Notification (AEN) system is concerned with managing the registration of events and handlers, and arranging for the queuing of raised events and the execution of the appropriate handlers in a pre-arranged context and priority. This is handled via the concept of a domain, which provides execution resources and a priority for the handling of events raised in that domain. Each domain may have several handlers for each event and may handle an event in a different maner from that of other domains. Each domain has a handler thread whose priority can be specified on domain creation, a set of mappings between events and handlers and a queue for raised events. Domains can be created and destroyed dynamically. Events are raised on a domain which may be specified directly or defaulted to the domain associated with the thread, allowing code to raise events without being aware of the target domain. The event is matched to its handler and put in a queue. This queue is processed, and the handlers called, within the context of the handler thread for the domain. Generic data can be passed into the event when it is raised and will be passed to the handler. The default domain for the handler thread is the domain itself, and event handlers are free to raise events themselves. In fact, since an event handler is run in a thread, there is no restriction on what a handler may do. There are global events. These have default handlers which can be changed dynamically on a system-wide basis. These handlers can also be overridden in a domain. There are also events which are domain specific. The creation and destruction of these domain specific

events and changing handlers for events is dynamic. However, changing the handler of an event will not effect any currently queued event raises. Global events can be raised across all domains in the system with a single request. In a given domain, any over-riding handler will be used instead of the global default handler for that event.

The nucleus provides a generic scheme to unify the majority of system resources and any number of user defined resources under a single mechanism. This single mechanism provides a consistent mechanism for integrating external resources within internal resources thereby simplifying the application level problems. Resources such as interface handles, vouchers, file descriptors (read and write) and channels can be managed in the following ways. Resources can be waited on either singly or in sets. Sets are created and maintained by the user. Waiting can be time bound or indefinite. A wait blocks the thread of control calling the wait (and only that thread) until either a resource becomes available (ready) or a time-out (if specified) occurs. Alternatively, events or call-backs can be registered to occur when a resource becomes available (ready). If an event is requested it will be raised in the domain of the thread registering the request when the resource becomes available. For a call-back the call-back function is queued to be executed on the stack of a special call-back thread. The resource module is driven by the application and the nucleus itself registering interest in the readiness of a resource or set of resources. The registered owner of the resource is then signalled by the resource manager to indicate interest in the resources state. If the resource becomes available (ready) then the owner of the resource informs the resource manager which then either wakes waiting threads, raises events or makes call-backs in order to inform interested parties that the resource has become ready. It is possible that the resource may not still be ready when an attempt is made to grab the resource, especially when many threads are waiting on a shared resource.

Referring now to figure 3, this shows in schematic form the pooling of tasks and their execution by threads within the nucleus.

In the arrangement of figure 3, a thread can execute only when it is associated with a task. Tasks are shared and re-used by threads in an application. Once a thread has completed execution the task it is executing on is released and can be used by another thread.

In figure 3 the tasks are represented by triangles and the threads by squares. The diagram shows the life-cycle of a task. A task starts in a free pool, i.e. it is unbound, then it is bound to a thread, the thread task pair are now ready to execute and enter a run-queue. The thread/task pairs in the run-queue are then scheduled to share the available processor time. When a thread has completed execution the task it is bound to is freed and returned to the free pool.

By separating logical and physical concurrency, threads and tasks, we can control the resources used to provide the concurrency in an application without affecting the logical concurrency of the application.

Our thread/task concurrency makes use of light-weight concurrency for its tasks, and provides an additional layer of logical concurrency on top. Additional benefits gained from this approach include:-

- inexpensive generation of logical concurrency;
- the physical resources required to provide concurrency can be limited, and controlled;
- physical resources can be pre-allocated removing the latency introduced by dynamic resource allocation;
- control can be exercised over the scheduling of the threads, without control being required over the scheduling of the tasks that will execute those threads; and
- customised thread scheduling algorithms can be used for different environments, i.e. deadline scheduling for multi-media.

This approach to providing concurrency has the advantage over conventional means of concurrency provision in that each thread does not require resources such as stack and register store to be allocated.

In order to provide light-weight concurrency, or multi-threading, the nucleus performs the following functions:-

- Prioritisation of concurrent activities - From the real-time work a need for certain application operations to be handled more urgently than others has introduced the need for prioritisation of application activities. This means that more urgent application activities are given a higher priority. Higher priority activities are then performed in preference to lower priority activities.

- Control over resources required for multi-threading.

- Synchronisation mechanisms - In order for a multi-threaded application to be written in a safe manner the concurrent components in the application must be able to synchronise their activity to guard critical regions of code against multiple concurrent usage.

- Incorporation/interworking with Kernel, or other, threads packages.

As discussed above, threads are the logical unit of execution in an application. However, a thread does not have the necessary resources for actual execution. This allows us to have many threads (potential units of execution) in an application, without the cost of providing the resources for each to be able to execute.

Provision of application concurrency requires mechanisms to control:

- thread management,
- thread memory management,
- thread synchronisation, and
- thread activity tracking.

Together these provide an environment for applications to successfully use multi-threading in a safe manner. This multi-threading environment is illustrated in figure 4.

Providing a multi-threaded environment needs functionality to provide:

- the allocation of thread resources, i.e. the creation/destruction of threads;

- the tracking of thread activities. In order to provide debugging and diagnostic information it is necessary to follow the flow of execution from parent threads to children, and also possibly from threads in one object to another, i.e. following a call from a client to a server and back again, and

- priority control for a thread.

This functionality is provided by a thread management object.

Multi-threaded programs need to allocate and free memory blocks in much the same way that single threaded applications do. However, the use of standard routines, such as UNIX *malloc* and *free*, is not sufficient. In a multi-threaded environment memory allocated using standard routines, and not explicitly freed by the application, will not b e freed when a thread terminates. Instead the memory will be freed when the capsule terminates. This causes memory leaks and will lead to application memory requirements growing over time. Functionality which allows memory to be allocated to specific threads, and to be freed when the thread completes execution is provided by a thread management object which is also responsible for providing portable reliable memory allocation and freeing routines for use by the application.

Provision of a multi-threaded environment introduces a need for functionality to enable threads to synchronise their activity. This includes:

- providing a mechanism to allow mutual exclusion for protection of shared data;

- the ability for a number of threads to synchronise their state after a period of concurrent execution;

- mechanisms to allow threads to generate regular events in the nucleus, for use in heartbeats, or periodic checking of the state of another capsule, and

- the ability to do time based waiting for events to be triggered by the nucleus.

The thread synchronisation primitives that can be used to provide the synchronisation functionality specified above include:

- Counting semaphores - used to protect resources, and critical regions by limiting the number of threads which can hold locks on the semaphore at any one time. Thus, three threads attempt to lock a semaphore with a limit of two threads, the third thread will be blocked until one of the first two threads return the lock they have taken.

- Mutexes and Condition Variables
  Mutexes are binary semaphores, i.e. counting semaphores with an active concurrency limit of one thread, with threads waiting to attain a lock on the mutex queued in priority order. Condition variables allow a number of threads to synchronise their activity upon the value of a shared state. Operations are provided to queue threads which are waiting for the value of the shared state to be modified by another thread so that they can continue execution and to signal that the value of the shared state has been changed so that other threads may now be able to continue execution.

- Event Counters and Sequencers
  Event counters hold a piece of state and threads can be queued waiting for the state of an event counter to reach a particular value. The value of the event counter is controlled by threads

making advance calls which cause the value to be increased by one. Sequencers provide threads with tickets whose values are guaranteed to be monotonically increasing. Together the two can be used to provide a guaranteed ordering of the execution of a number of threads, and can ensure that only a given number of threads are active at any one time.

Providing threads with time based synchronisation primitives requires the introduction of timers. Timers are structures which trigger events at a given relative or absolute, time. A timer has an action associated with it. The action will be scheduled to execute as soon after the expiry time is reached as possible.

Task Binding

Task binding is the mechanism by which waiting threads are associated with available tasks. This is effected via a task binding algorithm which address the following problems:

- A number of task pools will be managed by the nucleus
- Each pool can contain an arbitrary number of tasks.
- Task stack sizes are configurable, different tasks within the same pool may have different stack sizes.
- Low and high watermarks can be specified for each pool. The high and low watermark concept is a task resource management issue. Instead of creating and configuring the maximum number of tasks that a capsule will require during initialisation, reserving the maximum possible memory that the system will use, we create only a proportion of the tasks initially. This proportion is called the low watermark and is a lower limit on the number of tasks which are actually part of a given pool. Further tasks can be dynamically created, up to the limit specified by the high watermark, either on demand or using a lookahead mechanism. When tasks are no longer

required they can be destroyed, freeing memory, as long as the low watermark is not breached. Dynamically creating, and destroying, tasks keeps the resources required by the nucleus down, at the cost of increased latency introduced by having to create tasks to execute threads during periods of high activity.

- Pools can be either:
  - reserved for use by threads whose priority is greater than a given value, or
  - reserved for use by an interface, or group of interfaces, within a capsule.

Figure 5 is a diagrammatic representation of the task pools. In the diagram the four task pools are: G for general use, I for use by interfaces instances X_1, X_2, Y_2 only, 2 for use by threads of priority greater than or equal to 3, and 3 for use by threads of priority greater than or equal to 5. So low priority threads use tasks from the general pool only, hence ensuring that low priority tasks to do hog all of the nucleus' tasks. Pool 1 is reserved for use by a specific set of interfaces, this ensures that a minimum service guarantee can be provided to those interfaces.

This design allows the task pools to be configured in a variety of ways. Task pools can be created and destroyed dynamically, allowing new capsules to ensure that they have sufficient resources to execute properly. High and low watermarks are not shown on the diagram but have the effect that as all the available tasks are consumed from a pool more will be created until that pool has reached its high watermark. As the tasks are freed they will be destroyed until the pool reaches its low watermark once more.

The design of the task pools is flexible, but if task pools are not used by the application little or nor overhead should be incurred. It is important in the task pool implementation to ensure that the costs of task pool flexibility are only incurred by those requiring them.

Task Scheduling

There may be several thread/task pairs bound and awaiting to execute in the nucleus at any one time. The nucleus must organise these activities in some ordered manner; this process is usually referred to as scheduling.

Task scheduling can be of two basic types:

- Non-Preemptive Scheduling - Once a task has begun execution it continues to execute until it chooses to yield execution to another task. The current nucleus implements a simple non-preemptive scheduler. This is simple to implement and allows the use of non re-entrant libraries. However, it lacks fairness in that a thread/task may hog the processor.

- Preemptive Scheduling - A running thread-task can be stopped and another thread/task allowed to run. This increases the fairness of the scheduling but requires that ll code be written in a re-entrant manner or non-re-entrant code be protected by critical regions. POSIX provides a preemptive scheduler.

In preemptive scheduling where threads have priorities, the thread that has the highest priority runs. If a thread with a higher priority becomes ready, then it should pre-empt any lower priority threads. With preemptive scheduling, a number of policies can be used to control when thread/task pairs are switched out and replaced by others.

- First In First Out (FIFO). The thread/task at the head of the highest priority queue runs to full completion, or until it yields execution, unless pre-empted by a higher priority thread.

- Round Robin (RR). The threads in the highest priority queue are time-sliced, i.e. they share the available processor resources. Each thread executes for a given time period, called a quantum, and is then pre-empted.

- Time-sharing. Thread/task pairs are given a quantum according to their priority, larger quantum for higher priority activities, and then all activities are time-sliced according to their quantum and not their priority.

When Kernel threads are not available, a scheduler must be provided by the nucleus. A light-weight, non-preemptive , priority based, round-robin-like scheduler is provided as the default scheduler because:

- a non-preemptive scheduler is easier to implement than a preemptive one,
  - it does not require re-entrant libraries to be written for the standard C and UNIX libraries,
  - it can be upgraded at a later date, providing the Operating system libraries that go with it are re-implemented or protected in some way.
- a priority based scheduler is essential in an environment where there is a need to prioritise activities and give those prioritised activities preferential treatment
- the algorithm will be round-robin-like in that the highest priority activities will be non-preemptive ly cycled. By using round-robin we ensure that the highest priority activities be given preference by the scheduler.

Due to the modular nature of the nucleus design, it will of course be possible for different schedulers to be used in place of the standard one provided. In order to ensure this, the interface and behaviour which a scheduler must provide will be carefully documented.

Where preemptive scheduling is provided b y the kernel, the nucleus will allow the application of the option of using it. Some preemptive

scheduling systems allow the user to control the quantum value and algorithm used by the scheduler. These, and other, nucleus configurables can be specified both statistically (at compile time) and dynamically (at run time). The nucleus will handle the configuration of the underlying system, where appropriate functionality exists, and resolution of any conflicts between different application requirements.

Multi-threading summary

In summary we have identified a need for a number of component objects of a multi-threading system. All of these objects must be available to provide multi-threading.

However two distinct scenarios have been identified.:

- Where kernel threads are available we need merely encapsulate them to provide the functionality of the ODS task object, see Figure 5. The nucleus is responsible for providing the thread objects, the ODS task object provides a wrapper around the kernel threads and the task binding algorithm.

  Figure 4 shows the objects comprising the multi-threading environment. The task binding algorithm is responsible for choosing the next thread to be bound on any available tasks. The task binding will be priority based, and the application will be offered the option, when applicable, of using the tasks in either preemptive or non-preemptive mode.

- The nucleus is responsible for providing all components of the multi-threading environment not provided by the kernel, i.e. if kernel threads are not provided the nucleus must supply thread, task, stack and scheduler.

- The default scheduling algorithm used in the nucleus will be priority based non-preemptive .

The task module will provide the application with the ability to control the task binding algorithm by providing a mechanism to allows tasks to be reserved for use by threads of an acceptably high priority level.

The system provides a deterministic environment in which additional components may be integrated as appropriate. The framework includes mechanisms for threads and tasks, comms stacks (including but not limited to RPC), resource management and asynchronous event notification. The use of these components allows applications to be constructed which run over multiple clients and serves and enable servers to themselves be clients of other servers. In particular, the system provides an integrated framework for distributed applications in a real time environment. Tasks and threads provide abstractions to simplify the programmers handling of concurrency whilst giving fine grained control of resource usage. The programmer has the flexibility to define maximum and minimum resource availability for fine-grained activities. LMP is a particular protocol stack which is suited to a range of tasks from journalling with an asymmetric flow of data but efficient retransmission characteristics, through to RPC systems such as the heartbeat protocol. Resource Management is an integrated way of enabling efficient coexistence of multiple resource types within a common model suitable for real-time application development. The model enables determinism, prioritisation and deadline scheduling for such resource utilisations Asynchronous event notification is a model for event handling which fits in with the environments above enabling applications programmers to handle them in a fully integrated manner. This preserves the deterministic characteristic in the presence of external systems.

CLAIMS:-

1. A communications network, including a plurality of switch elements, a controller for the switch elements, and a system manager, wherein the controller comprises a main computer system and a reserve computer system, there being means for switching from the main computer system to the reserve computer system in the event of failure of the main computer system, wherein the main computer system has means for communicating state changes in the form of a message stream to the reserve computer system whereby to update the reserve system with the current network state, and wherein the main computer system is arranged to provide status information to the reserve computer system at regular intervals whereby to detect an in-service failure of the main computer system.

2. A communications network providing client access to a plurality of servers, the network including a plurality of capsules each containing client and server objects and each incorporating a nucleus supporting a plurality of client/server interfaces, wherein each said nucleus includes means for pooling tasks, means for providing threads and for assigning those threads to respective tasks, and means for binding an assigned thread to its respective task during the execution of that task.

3. A communications network as claimed in claim 1, and incorporating means for asynchronous notification of system events.

4. A method of controlling a telecommunications network including a plurality of switch elements, a controller for the switch elements, and a system manager, wherein the controller comprises a main computer system and a reserve computer system, the method including communicating state changes in the main computer system to the reserve computer system whereby to update the reserve system with the current network state, providing status information from the main computer system to the reserve computer system at regular intervals whereby to detect in service failure of the main computer system, and

substituting the main computer system with the reserve computer system in the event of a detected failure of the main system.

Fig.1. 1/3

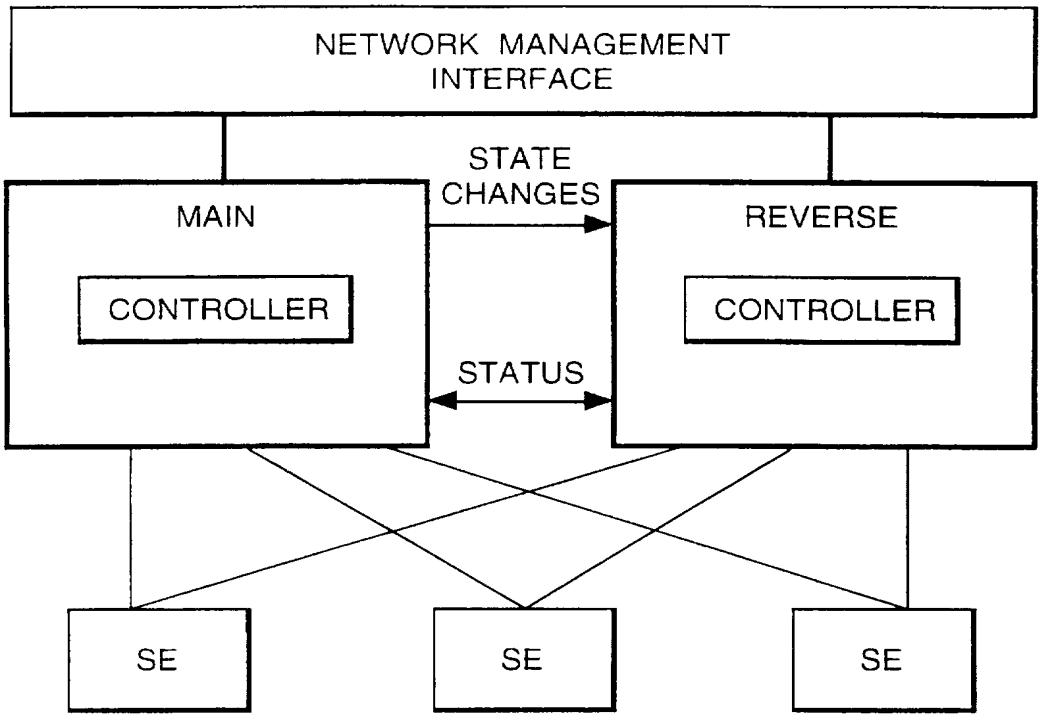

NETWORK MANAGEMENT INTERFACE
STATE CHANGES
MAIN
CONTROLLER
REVERSE
CONTROLLER
STATUS
SE
SE
SE

Fig.5.

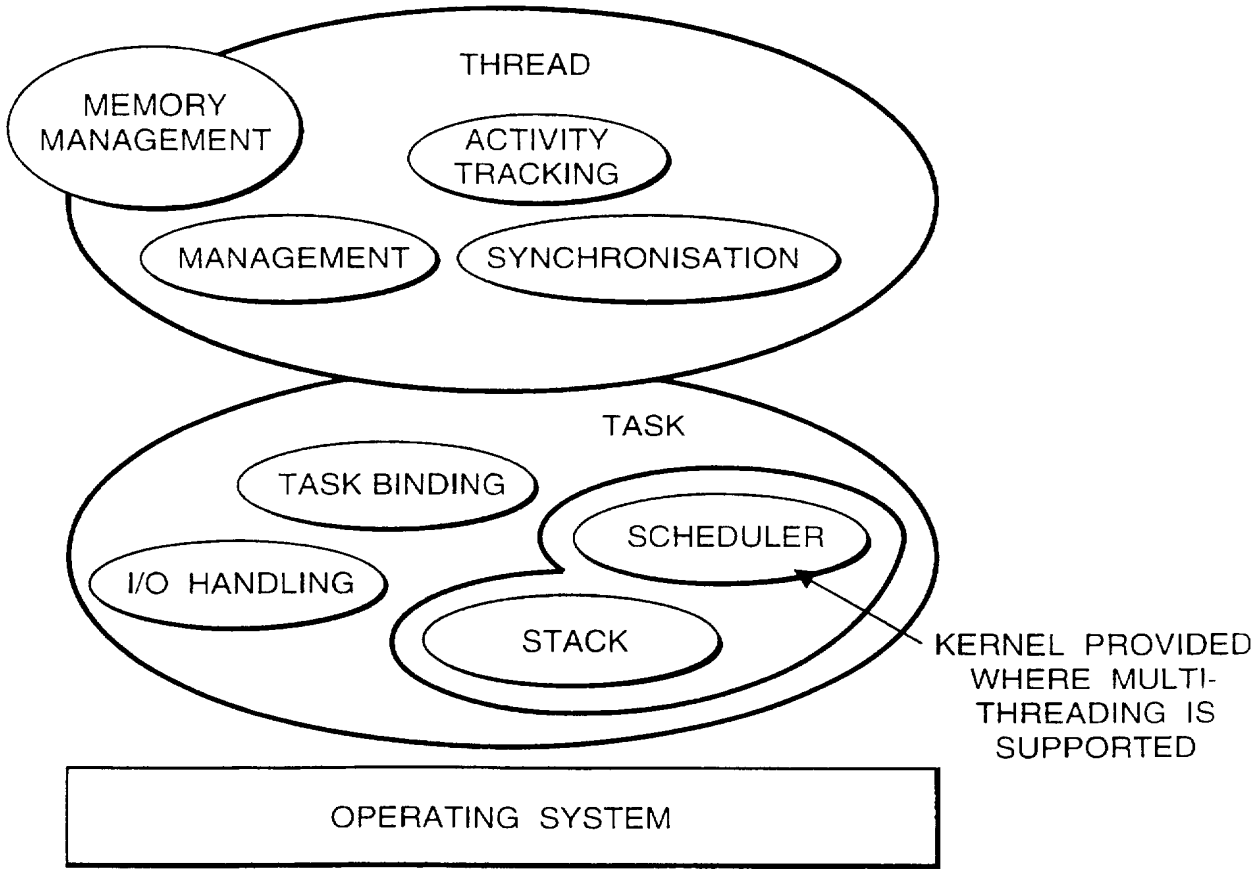

THREAD
MEMORY MANAGEMENT
ACTIVITY TRACKING
MANAGEMENT
SYNCHRONISATION
TASK
TASK BINDING
SCHEDULER
I/O HANDLING
STACK
KERNEL PROVIDED WHERE MULTI-THREADING IS SUPPORTED
OPERATING SYSTEM

Fig.2.

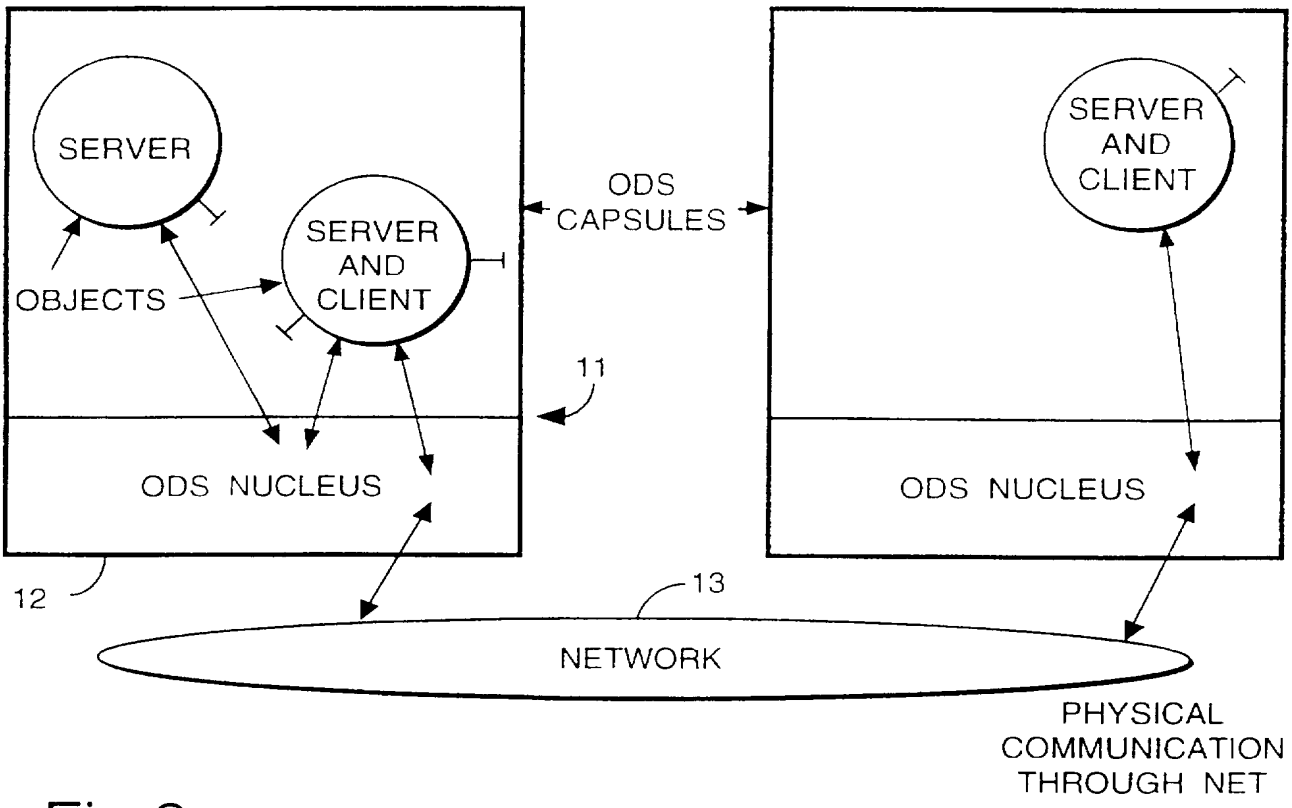

SERVER
SERVER AND CLIENT
OBJECTS
ODS CAPSULES
SERVER AND CLIENT
11
ODS NUCLEUS
ODS NUCLEUS
12
13
NETWORK
PHYSICAL COMMUNICATION THROUGH NET

Fig.3.

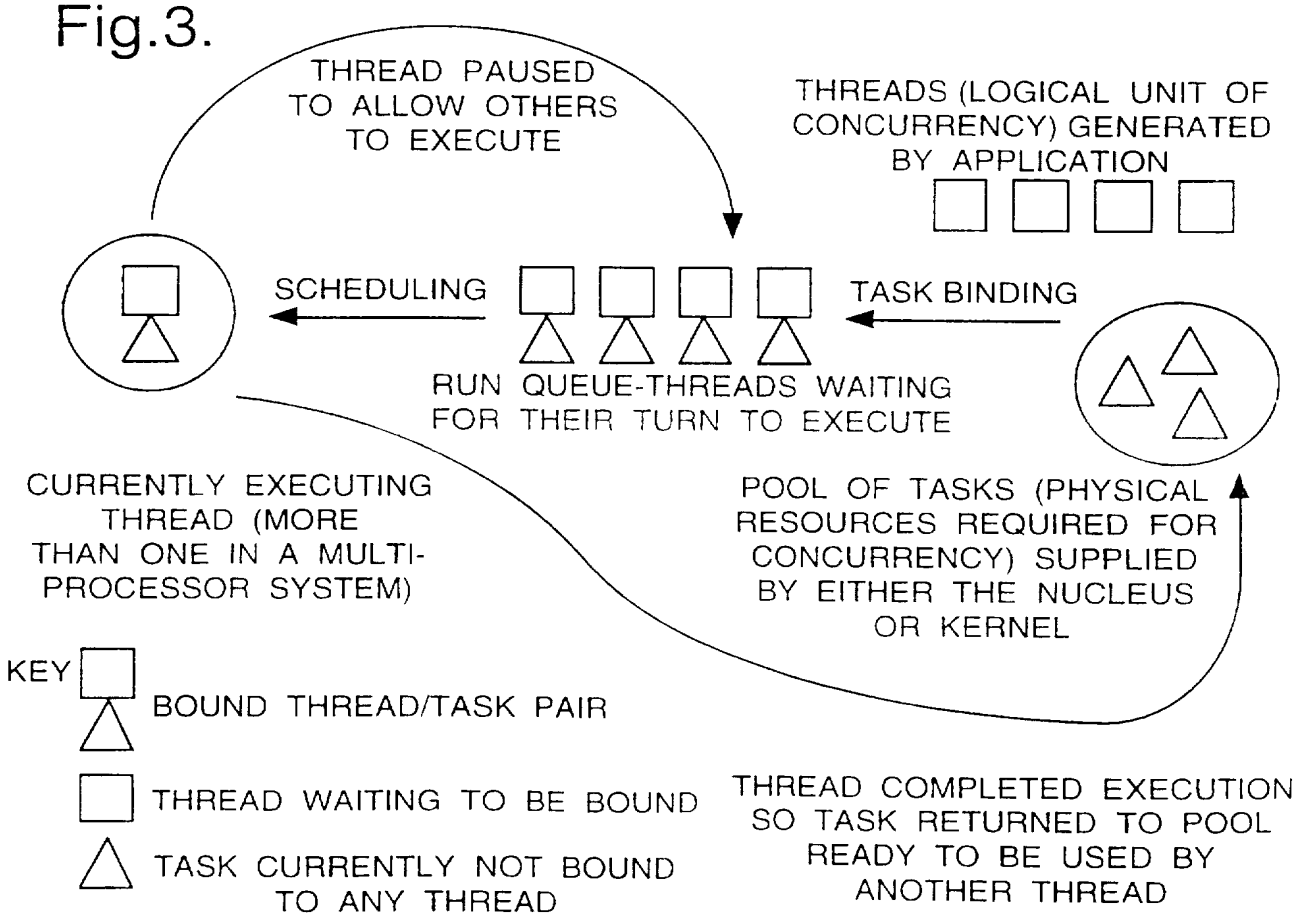
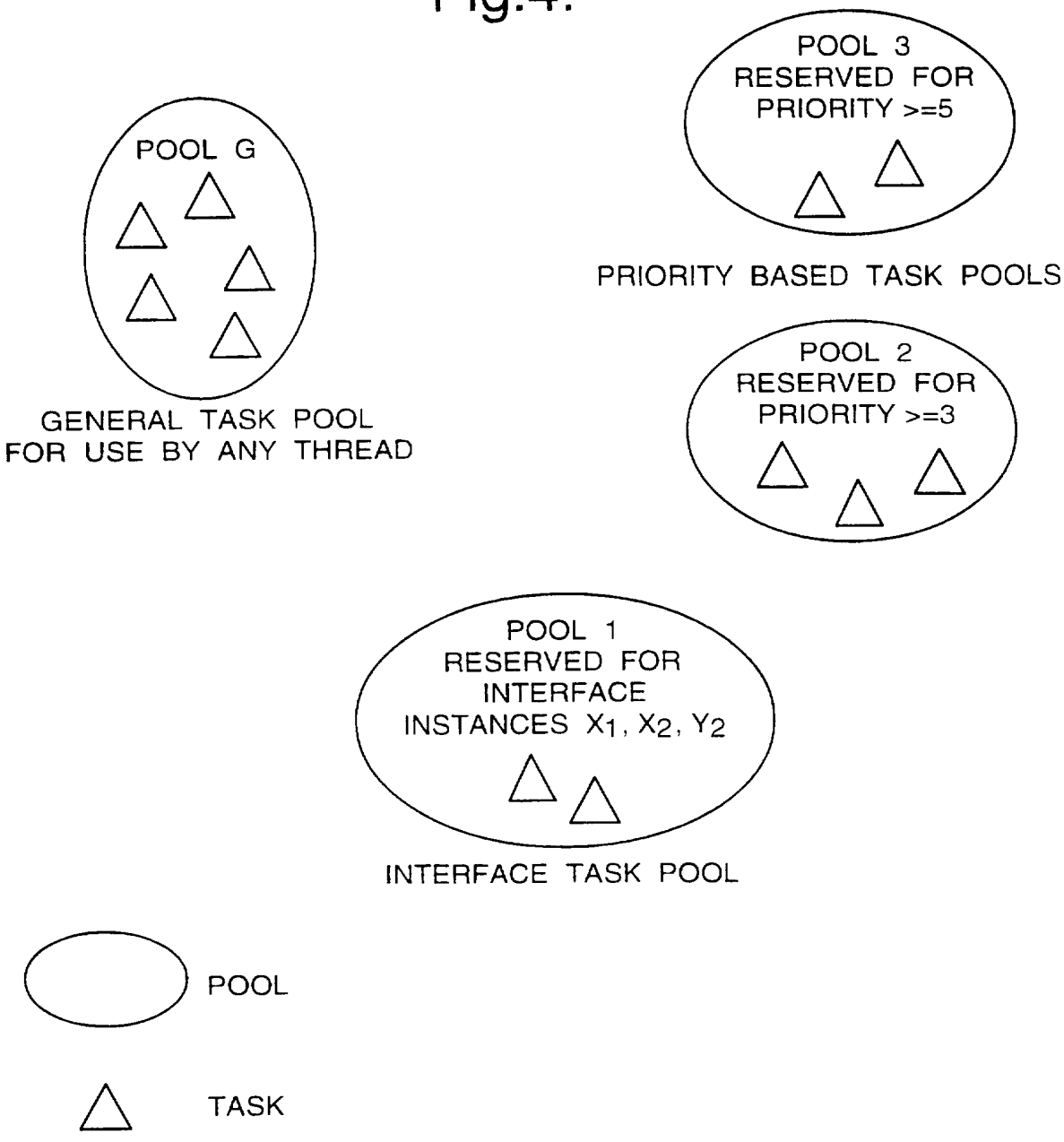

THREAD PAUSED TO ALLOW OTHERS TO EXECUTE
THREADS (LOGICAL UNIT OF CONCURRENCY) GENERATED BY APPLICATION
SCHEDULING
TASK BINDING
RUN QUEUE-THREADS WAITING FOR THEIR TURN TO EXECUTE
CURRENTLY EXECUTING THREAD (MORE THAN ONE IN A MULTI-PROCESSOR SYSTEM)
POOL OF TASKS (PHYSICAL RESOURCES REQUIRED FOR CONCURRENCY) SUPPLIED BY EITHER THE NUCLEUS OR KERNEL
KEY
BOUND THREAD/TASK PAIR
THREAD WAITING TO BE BOUND
TASK CURRENTLY NOT BOUND TO ANY THREAD
THREAD COMPLETED EXECUTION SO TASK RETURNED TO POOL READY TO BE USED BY ANOTHER THREAD

Fig.4.


POOL G
GENERAL TASK POOL
FOR USE BY ANY THREAD
POOL 3
RESERVED FOR
PRIORITY >=5
POOL 2
RESERVED FOR
PRIORITY >=3
PRIORITY BASED TASK POOLS
POOL 1
RESERVED FOR
INTERFACE
INSTANCES X1, X2, Y2
INTERFACE TASK POOL
POOL
TASK